Electronic Notes in Theoretical Computer Science **14** (1998)
URL: http://www.elsevier.nl/locate/entcs/volume14.html 8 pages

An Incremental Model for Concurrent Objects

Sergio E. R. de Carvalho[1] & Toacy C. de Oliveirae

Laboratório de Métodos Formais
Departamento de Informática
Pontificia Universidade Católica do Rio de Janeiro
Rua Marqués de São Vicente 225
Rio de Janeiro, RJ, 22453-900, Brazil
Email: sergio,toacy@inf.puc-rio.br

Abstract

Numerous models have been proposed for concurrent objects, and their implementations in programming languages have been discussed. This paper presents an incremental model for object-oriented concurrency, starting out with a set of basic assumptions, which try to minimize user participation in designing concurrent systems. Greater demands for concurrency are met by incremental model modifications, during design and execution. The model itself is object-oriented, and can be considered as part of a basic framework for concurrent object-oriented applications.

1 Introduction

This paper presents the model for concurrent objects being adopted in the ARTS software paradigm. ARTS main goal is the definition and implementation of a set of languages, models, theories and tools for the method-independent development of real-time object-oriented systems. The ability to express concurrency (and consequently synchronization and communication) is essential to real-time systems. Considerable productivity increases can be expected with the use of development systems that simplify the expression of concurrency.

The basic architecture of ARTS consists of two planes. In the user plane software designers model object-oriented systems, for example via annotated

[1] This work was supported by CNPq, and partially sponsored by the project ARTS (formal Approach to Real-Time Systems development) being carried out by The Laboratory of Formal Methods, Department of Informatics, Pontifcia Universidade Católica do Rio de Janeiro, Brazil, and its research associates, under contract to Equitel S.A. Equipamentos e Sistemas de Telecomunica"cões.

class relationship, object communication, state behavior and scenario diagrams. In the formal plane, unseen to users, steps taken during design are verified, guaranteeing design consistency. Underlying platforms provide user-formal mappings. Designs are automatically coded in a Design Description Language [4], a very high-level language representing the object-oriented discourse adopted, and then automatically transformed onto C++ code.

ARTS applications can be modeled as instances of a framework consisting of a main object (a broker [3]) responsible for creating and initializing other objects in the application, including system interfaces, business objects, and object wrappers for platform resources, such as real-time clocks and thread pools. This is a pure object-oriented framework, without external referencing environments. Objects contain as attributes components responsible for handling their own user interfaces, business rules, persistence, and so on [13]. This modeling is not exclusive; other object-oriented architectures can be used in the ARTS paradigm.

Section 2 describes the design criteria we used to model concurrency. Section 3 presents work related to the construction of our model. Section 4 introduces features already available in the users plane, which may suggest concurrency to underlying platforms, such as run-time libraries and translators. Section 5 introduces the basic concurrency model adopted, and section 6 presents our conclusions and suggestions for further work.

2 Design Criteria

The development of concurrent systems is hard, imposing severe strains on designers. For this reason, to introduce concurrency in sequential ARTS, we adopted simplicity as our main concern: the visual and textual design languages should be minimally extended with concurrency features, and as much functionality as possible should be automatically obtained from underlying platforms and libraries, for example. In this way ARTS users should not be overly concerned with concurrency implementation issues. For example, issues relating to message queues and thread allocation concern users only indirectly, or not at all, as will be seen below.

Our second concurrency design criteria was the expressive power of the mechanisms adopted, as related to our application domain. For example, decisions had to be taken with respect to the level of internal concurrency in objects, a contrasting force to the preservation of object state, and to the expression of synchronism.

Finally, we already had a sequential object-oriented discourse, realized both in a graphical design space and in a description language, with built-in automatic C++ code generation. A few features of this discourse, already addressing invocation and synchronism, for example, constrained the introduction of new concurrency related aspects.

3 Related work

Several approaches are being used to relate concurrency and object-orientation. Proposals, results and desiderata are described for example in [11]. Solutions can be found at the language level, for example via special constructs added to a sequential language model [9], or via classes modeling concurrency concepts, as in Java; they are also present in development methods, as in Syntropy [6], and more recently design patterns have been proposed to model concurrency in the object paradigm [12,7]. This section briefly presents those approaches that had the greatest influence in our model, introducing the basic decisions taken in its definition.

There are two basic strategies for handling concurrency in object-orientation: execution environments (threads) can be orthogonal or integrated to objects [2]. In the orthogonal approach, a thread runs independently of objects, and may change the state of several objects during its life-cycle; also, operations are usually executed in the clients thread. In the integrated approach, threads are strongly connected to objects. Actor languages [1] favor this strategy. In its more general form, for each message received a new execution environment may be created, facilitating asynchronism, since the client is not blocked while the message is being processed in the servers thread. Our concurrency model uses basically the integrated approach.

Another basic concern in handling concurrency relates to the presence of active and passive objects. Active objects control their own destiny, having their own threads; passive objects execute on active objects threads. Active objects may also have an incoming message queue, and this facilitates an implementation of asynchronism: messages will be received and processed, but clients are not blocked in waiting. Systems may have objects of both kinds, and the distinction is sometimes left to programmers, via language constructs, as for example the **separate** keyword proposed for Eiffel [9]. Our model supports both active and passive objects. In many cases underlying platforms are able to select those that may have threads from those that do not need them.

The Threaded Active Object model [10] uses guarded active objects to provide concurrency and synchronization. Guards are associated with operations via a new **guard** construct, and are supported by deontic logic operators that increase the expressive power of synchronization constraints, and avoid the inheritance anomaly problem [14]. A restricted guard mechanism is also part of our model. Design patterns are being proposed as concurrency models. Thread per Request, Thread per Object and Thread Pool [12,7] are examples of patterns that summarize much of the necessary functionality for implementing concurrency in object-oriented systems. Using Thread per Request, to each message received there corresponds a new thread allocated to the receiving object, which guarantees a high degree of concurrency. Using Thread per Object, active objects receive a thread each, which decreases the concur-

rency level of applications, but simplifies the specification of synchronization constraints. Finally, a Thread Pool is a high level object (or wrapper) responsible for providing threads when necessary, and for scheduling execution when threads are not available. Our model uses concepts described in Thread per Object and Thread Pool.

4 Context

Already available, both in ARTS visual design tool and design language, are useful facilities for modeling concurrency, as for example object applicable operations that can be defined in classes with different synchronization and communication semantics, distinguished by different keywords:

- procedures block the client, can have input and output parameters, and can return an object;
- asynchronous message handlers do not block the client, and can have only input parameters;
- handshake message handlers impose a wait on the client, until the message is received, and can have only input parameters;
- future message handlers do not block the client, and allow the client to self-block in some future point in time, to receive output parameters from the server;
- iterators (used only to control loops) and coroutines allow objects to cooperate, while maintaining their own environments;
- exception handlers allow the correction of object states, so execution can proceed.

The selection of an execution mode for an operation, typically made at the graphical design level, must take into account the synchronization and communication needs for the operation. If an object needs a service rendered and cannot proceed until receiving results from this service, then it must block itself until the service terminates. In this case the semantics of the service should be that of a procedure. If, however, synchronization can be relaxed, for example via asynchronous messages, the degree of concurrency can be increased: a smart set of platforms can generate a new execution environment (or activate an existing one) and have both client and server executing concurrently.

Opportunities for these optimizations occur initially at translation-time, when objects modeled by classes offering more than the procedural behavior are declared. Such objects are candidates for concurrency, and may receive their own execution environment if in fact called upon to handle a message or to execute cooperatively. In the first case, various execution environments expedite the overall execution; in the latter, they simplify the necessary retention of local environments and resume addresses.

5 The Concurrency Model

We use the reflexive approach to explain the adjustments an object may go through during its lifetime in an ARTS application, from design to execution. In ARTS each application object is controlled by a manager, in charge of handling several aspects relating to its behavior, as for example message receiving, guard evaluation, and operation scheduling. This manager is also responsible for obtaining system resources that may be needed by its application object, such as time stamps and threads. Our basic idea is that this manager acquires structure and behavior as a function of:

- the declaration of the application object;
- the declaration of the application objects class;
- polymorphic transformations the application object suffers at execution time.

In the remainder of this section we detail this controlled behavior. Note that even during design, an application objects manager is taking shape; the first two items above relate to declarations, which can change during the visual construction of design diagrams. At design completion, the initial manager for each application object has a well-defined state, in which it is created at execution time. Application objects and their managers are created simultaneously.

Note also that in this model the creation of managers is completely transparent to designers, being the responsibility of the CASE tool where design takes place, the internal representation of designs, and their transformations to code. In this way designers can concentrate on the task at hand, modeling with less concern for low- level concurrency aspects.

The basic organization.

The manager of an application object begins its design life- time when the objects declaration is processed. At this time it acquires an initial state, containing as components a descriptor and a message queue, both to be used at execution time. This initial state may change, with the addition of new components, as the declaration of the application objects class is examined.

The descriptor will record, at execution-time, the application objects current class, and does not change during design. The message queue will store the operation requests the application object receives, and may change at design-time, for example if real-time limits are associated to an operation.

Adding guards.

Class defined operations may have guards, similar to those proposed in [10], which can consult, via the manager, the objects state and message queue. To model operation guards, managers of application objects of that class acquire guards vectors (still at design-time). Note that in our design tool, guards may

be associated to operations of a given class after several objects of this class (and their managers) have been processed; once a guard is defined, all objects (and managers) of the corresponding class must be updated.

Guards vectors are carried over to execution-time, and, as suggested in Syntropy [6], are updated during operation epilogues.

Adding threads.

If application objects are modeled by classes containing operations other than procedures, they are potentially active. If, at execution-time, they must satisfy non-procedural requests, their managers also need an execution stack, to fulfill the expected thread functionality, thus increasing concurrency. In this case managers are updated at execution-time.

Compiler optimizations, run-time systems and internal libraries help deciding which objects should actually be active. As a starting point, objects modeled by classes possessing only procedures as exported operations can be modeled as client stack parasites, due to the synchronism procedures impose. Compiler optimizations can also decide which operations may change the state of an object, and thus decide which operations should be executed in mutual exclusion, promoting the one writer, many readers serialization protocol [6].

Adding real-time.

During design, users may impose lower and upper bounds on operation execution times [5]. In this case the message queue must contain, for each real-time operation requested, its lower and upper bounds and a time stamp, obtained from the real-time clock via a managers request to the main object. This stamp corresponds to the time the request enters the servers message queue. This is directly accommodated in ARTS, since real-time messages are modeled by a subclass of class Message, and can thus be present in the message queue.

This new information is translated into operation pre- and post-conditions, which evaluate the initial execution time and the operation conclusion time, respectively, again based on time stamps. Exceptions are raised if these times are not met.

Adding priorities and exception handling.

In real-time systems operation requests may have different priorities, and an operation being executed on behalf of a client may even be interrupted. To handle priorities the functionality of managers must be upgraded, with the addition of a scheduler.

To handle interruptions, exceptions are raised. As proposed in [9], the interruption of an operation causes an exception in the client, since the state of the server may be modified by the intervening operation. The clients exception handler is thus invoked, taking appropriate actions.

6 Conclusions

This paper informally describes a concurrency model for the ARTS paradigm. This model uses the polymorphic transformation of manager objects to achieve a certain degree of concurrency among application objects. Managers are responsible for handling all concurrency aspects relating to application objects, considered as active, single-threaded objects.

In designing a model, attention has to be given to certain factors, or forces, which act on the domain for which the model is being constructed. Such forces are usually contrasting, and a good design is obtained by balancing out the most relevant forces considered. To model concurrency, designer involvement and adherence to the existing discourse were the main forces considered.

Designer involvement. This we wanted to minimize. The design of concurrent systems is hard, as evidenced for example by the ever growing number of design patterns addressing concurrency. Pattern conferences usually set aside concurrency sessions, and even in modern programming systems such as Java, collections of patterns are recommended in modeling concurrency [8]. Such patterns propose to alleviate the expression of concurrency, providing proven solutions to small scale problems.

Adherence to the existing discourse. This we wanted to maximize. As mentioned, we had available an object-oriented model in which class operations could be expressed with different semantics, and where automatic transformations already took advantage of the threading facilities of Windows NT, the selected implementation platform. Moreover, the visual design tool already contained state behavior and class relationship diagrams, which could be used in modeling concurrency.

Balancing out the two forces above had as consequences an increase in the ease of design and in the safety of concurrent systems, and a corresponding decrease on the degree of concurrency that can be expressed. In fact, the current model does not support intra-object concurrency, nor does it provide replication. In contrast, synchronization and communication between concurrent objects is simplified, being expressed at a higher level than usual, via the different semantics that can be attached to operations at design time.

References

[1] Agha, G. "A Model of Concurrent Computation in Distributed Systems," MIT Press, 1986.

[2] Atkinson, C. , "Object-Oriented Reuse, Concurrency and Distribution: an Ada based Approach," ACM Press, 1991.

[3] Buschmann, F. , R. Meunier, H. Rohnert, P. Sommerlad, M. Stahl, "Pattern - Oriented Software Architecture," Wiley, 1996.

[4] Carvalho, S., *The design description Language,* MCC 29/97, Departamento de Informtica, PUC-Rio, September 97, available from `ftp://ftp.inf.puc-rio.br/ftp/pub/docs/techreports`.

[5] Carvalho, S., J. Fiadeiro, H. Hausler, *A formal approach to real-time object-oriented software,* Proceedings WRTP97, Lyon, France, September 1997, pp. 91-96.

[6] Cook, S., J. Daniels, "Designing Object Systems," Prentice-Hall, 1994.

[7] Grasso, E., *Synchronizer - an object behavioral pattern for concurrent programming,* Proceedings EuroPLoP 97, Irsee, Germany, July 1997, pp.153-164.

[8] Lea, D., "Concurrent Programming in Java," Addison Wesley, 1997.

[9] Meyer, B., *Systematic concurrent object-oriented programming,* CACM, September 1993, Vol. **36**, No. 9.

[10] Mitchell, S. E. and A. J. Wellings, *Synchronization, concurrent object-oriented programming and the inheritance anomaly,* Computer Languages, Vol. **22**, No. 1, 1996.

[11] Papathomas, M., *Concurrency in object-oriented programming languages,* in: "Object-Oriented Software Composition," O. Nierstrasz and D. Tsichritzis, eds., Prentice Hall, 1995.

[12] Petriu, D. and G. Somadder, *A pattern language for improving the capacity of layered client/server systems with multi-threaded servers,* Proceedings EuroPLoP 97, Irsee, Germany, July 1997, pp.179-190.

[13] Vianna e Silva, M. , S. Carvalho, J. Kapson, *Patterns for layered object-oriented applications,* Proceedings EuroPLoP 97, Irsee, Germany, July 1997, pp.85-94.

[14] Tomlinson, C., W. Kim, M. Scheevel, V. Singh, B. Will, G. Agha, *Rosette: an object-oriented concurrent systems architecture,* ACM SIGPLAN Notices **24**(4), PP. 91-93, 1989.